THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

A Rational Reconstruction and Extension of Recursion Analysis

Citation for published version:
Bundy, A, van Harmelen, F, Hesketh, J, Smaill, A & Stevens, A 1989, A Rational Reconstruction and Extension of Recursion Analysis. in *Proceedings of the 11th International Joint Conference on Artificial Intelligence - IJCAI 1989.* <http://ijcai.org/Past%20Proceedings/IJCAI-89-VOL1/CONTENT/content.htm>

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 11th International Joint Conference on Artificial Intelligence - IJCAI 1989

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

OPEN ACCESS

Download date: 15. Jan. 2021

A Rational Reconstruction and Extension of Recursion Analysis *

Alan Bundy, Frank van Harmelen, Jane Hesketh
Alan Smaill, and Andrew Stevens
Department of Artificial Intelligence, University of Edinburgh,
80 South Bridge, Edinburgh, EH1 1HN, Scotland

Abstract

The focus of this paper is the technique of *recur8\on analysis.* Recursion analysis is used by the Boyer-Moore Theorem Prover to choose an appropriate induction schema and variable to prove theorems by mathematical induction. A rational reconstruction of recursion analysis is outlined, using the technique of *proof plans.* This rational reconstruction suggests an extension of recursion analysis which frees the induction suggestion from the forms of recursion found in the conjecture. Preliminary results are reported of the automation of this rational reconstruction and extension using the CLAM-Oyster system.

1 Introduction

The work described in this paper is part of a project to explore the use of *proof plans* for the automatic guidance of mathematical proofs. In particular, we are developing proof plans for the proofs by mathematical induction that are required in the automatic synthesis of computer programs from their specifications. Given a conjecture, the CLAM plan formation program constructs a proof plan to solve it from existing proof tactics. This proof plan is then used to guide the Oyster proof development system, [Horn, 1988], in a proof of the conjecture. Oyster is a Prolog re-implementation of Nuprl, [Constable *et* a/., 1986]. They are both proof checkers for Intuitionistic Type Theory, a logic based on the work of Martin-Lof.

Oyster reasons backwards from the conjecture it is trying to prove, using a sequent calculus formalism which includes rules of inference for mathematical induction. The search for a proof must be guided either by a human user or by a Prolog program called a *tactic.* The Oyster search space is very big, even by theorem proving standards. There are hundreds of rules of inference, many of which have an infinite branching rate. So careful search is very important if a combinatorial explosion is to be avoided.

Our aim is to develop a collection of powerful, heuristic tactics that will guide as much of the search for a proof as possible, thus relieving the human user of a tedious and complex burden. These tactics need to be applied flexibly in order to maximise Oyster's chances of proving each theorem. Proof plans provide this flexible application. Each tactic is partially specified by a *method,* which is a description, in a meta-logic, of the preconditions and effects of the tactic. CLAM reasons with these methods to build proof plans. A proof plan for a conjecture defines a tactic tailor-made to generate a proof of that conjecture.

The state of the art in inductive theorem proving is the Boyer-Moore Theorem Prover, [Boyer and Moore, 1979] (henceforth BMTP). It is, thus, natural for us to try and represent the heuristics embedded in the BMTP as Oyster tactics. [Bundy, 1988] contains an analysis of some of these heuristics. We have used this analysis to implement a number of Oyster tactics for inductive proofs and have successfully tested them on some simple theorems drawn from the literature, [Bundy *et* a/., 1988].

2 Recursion Analysis

Recursion analysis is the name we have given to the process, embedded in BMTP, of analysing the recursive structure of a conjecture and deciding what form of induction to use to prove it. This choice is critical to the success of the proof attempt. Recursion analysis picks one out of several *induction suggestions.* An induction suggestion consists of an induction schema and some induction variables[1]. The form of the recursive functions contained in the conjecture is used to construct *raw induction suggestions.* These are then combined together into a *final induction suggestion,* which is the one used in the proof. In this section we will explain the rational reconstruction of recursion analysis given in [Stevens, 1988]. In later sections we will see how to realise this rational reconstruction within proof plans and how to extend it.

We can best explain how recursion analysis works by example. Consider the conjecture:

$$\begin{aligned}&\forall x : pnat, \forall y : pnat.\\ &\{even(x) \wedge even(y) \quad \rightarrow \quad even(x+y)\} \qquad (1)\end{aligned}$$

where the recursive definitions of -f and *even* are given in the top left hand corner of figure 1. Formulae of the

*The research reported in this paper was supported by SERC grant GR/E/44598, Alvey/SERC grant GR/D/44270, an SERC Senior Fellowship to the first author and an SERC postgraduate studentship to the last author. We are grateful to the other members of the mathematical reasoning group at Edinburgh for many useful discussions and to an anonymous IJCAI-89 referee for extensive feedback.

however, in subsequent sections we will restrict ourselves to single variable inductions.

form *X:T* are to be read as "X is of type T", and *pnat* is the type of Peano natural numbers. Note that the definition of + steps down in single steps, whereas the definition of *even* steps down in double steps. The number of steps is determined by the term occurring in the recursive argument position in the head of the step formula of the definition. These terms are emphasised with an underbrace in the definitions of + and *even* in figure 1. We will call them *recursion terms.* The recursive function in the body of the definition is emphasised by an overbrace.

To each recursion schema there corresponds a dual induction schema. For instance, the induction schema dual to 1 step recursion is:

$$\frac{\Gamma \vdash P(0) \qquad \Gamma, x':pnat \quad P(x') \vdash P(\underbrace{s(x')})}{\Gamma, x:pnat \ \vdash P(x)}$$

whereas, that dual to 2 step recursion is:

$$\frac{\Gamma \vdash P(0) \quad \Gamma \vdash P(s(0)) \quad \Gamma, x':pnat\ , P(x') \vdash P(\underbrace{s(s(x'))})}{\Gamma, x:pnat \ \vdash P(x)} \tag{2}$$

where *P(x)* ranges schematically over formulae which contain x[2]. To emphasise the duality between induction and recursion, the *induction terms* of each schema are also underbraced. Note that we have renamed the induction variable, x, to x' in the induction hypothesis and conclusion, *cf.* the renaming of *x* to *x'* in figure 1. This renaming is necessary for soundness, since *T* may contain x.

Recursion analysis locates the recursive functions in the conjecture. Each occurrence of a recursive function, *F*[3], with a variable, X, in its recursive argument position, gives rise to a raw induction suggestion. The induction variable suggested is *X.* The induction schema suggested is the one dual to the form of recursion used to define *F.* Applied to the conjecture 1 this produces the raw induction suggestions given in table 1.

X	F	Schema	Recursion Term	Status
x	*even*	2 step	$s(s(x))$	unflawed
y	*even*	2 step	$s(s(y))$	flawed
x	$+$	1 step	$s(x)$	subsumed

Table 1: Induction Suggestions for the Even+ Example

Note that each occurrence of *x* gives rise to a raw induction suggestion. Unfortunately, the two occurrences suggest two different induction schemata: a 2 step and 1 step schema. Fortunately, the 2 step schema *subsumes* the 1 step schema. This is because the recursion term of the 2 step schema consists of repeated nestings of the recursion term of the 1 step schema. Thus the 2 step, *x* schema is adopted as the single schema for x, replacing both itself and the 1 step, *x* schema. In general, there may not be an already suggested schema for a variable that subsumes all the others, but it may be possible to find a new schema that does. Stevens' recursion analysis, [Stevens, 1988], finds such all-subsuming schemata by *merging* one or more raw induction suggestions. For instance, a 2 step and 3 step induction would be merged to give a 6 step schema that subsumed both[4].

The end result of merging is a set of suggestions covering every distinct (i.e. non-subsumed) induction schema that can be derived from the raw induction schemata. In our example we would have a 2 step schema for *x* and a 2 step schema for y. All that remains is to choose which one should become the final induction suggestion. Note that the 2 step, *x* schema will produce an induction conclusion with the term *s(s(x'))* replacing each occurrence of x, and that, by design, a recursive definition will match the term immediately dominating each of these occurrences in the induction conclusion, namely even(s(s(x'))) and s(s(x')) + y. The same is not true of the 2 step, y schema. The second occurrence of y did not play a role in formation of this induction suggestion, and no recursive definition will match the term x + *s($(y'))* which it gives rise to in the induction conclusion. The replacement of the second occurrence of y with s(s(y')) is regarded as *unsuitable.* Boyer and Moore classify such suggestions as *flawed*[5]. Flawed suggestions are rejected if any unflawed ones remain, so that in the example above, the 2 step, x suggestion is the one that is finally chosen. In the event of a tie the induction subsuming the largest number of raw suggestions is chosen.

BMTP and [Stevens, 1988] use recursion analysis to construct induction schemata at run time, and hence must prove them well-founded after construction. We have not reconstructed this aspect of recursion analysis, and so do not discuss it further here. Our induction suggestions must be formed from those schemata for which Oyster currently has induction rules of inference.

The BMTP only uses the final induction suggestion generated by recursion analysis. Note that this makes its choice dependent on the forms of recursion found in the conjecture. For instance, it could not prove the classic version of the prime factorisation theorem[6]. This uses a form of induction in which the induction term is p x x', where p is a prime number. No *p* x *x'* recursion occurs in the theorem itself, or can be constructed by merging recursions in the theorem.

Nevertheless, recursion analysis is extraordinarily successful. For many simple theorems, the final form of induction it suggests leads to a proof. Why is this?

[2] But it may also contain other variables, and is not necessarily unary.

[3] *F* is a meta-variable ranging over object-level functions. We use the conventions that: object-level variables are represented as meta-level constants, only meta-level variables start with an upper case letter, and the word 'function' is used in the lambda calculus sense to include predicates and connectives.

[4] The original BMTP deals with subsumption as a situation distinct from (as opposed to subsumed by) merging. One consequence is that it is unable to deal with the 2/3 step situation and others like it.

[5] See [Boyer and Moore, 1979] or [Stevens, 1988] for a full definition of unsuitable replacements and flawed suggestions.

[6] BMTP does prove a non-classic, 'verification' version of this theorem which, in the place of an existentially quantified variable, includes a function, defined by *p* x *x'* recursion, which returns the prime factors of a natural number. In the classic version (see §7 below) these factors are merely asserted to exist. Because the classic version includes an existential quantifier, it cannot even be stated in the BMTP logic.

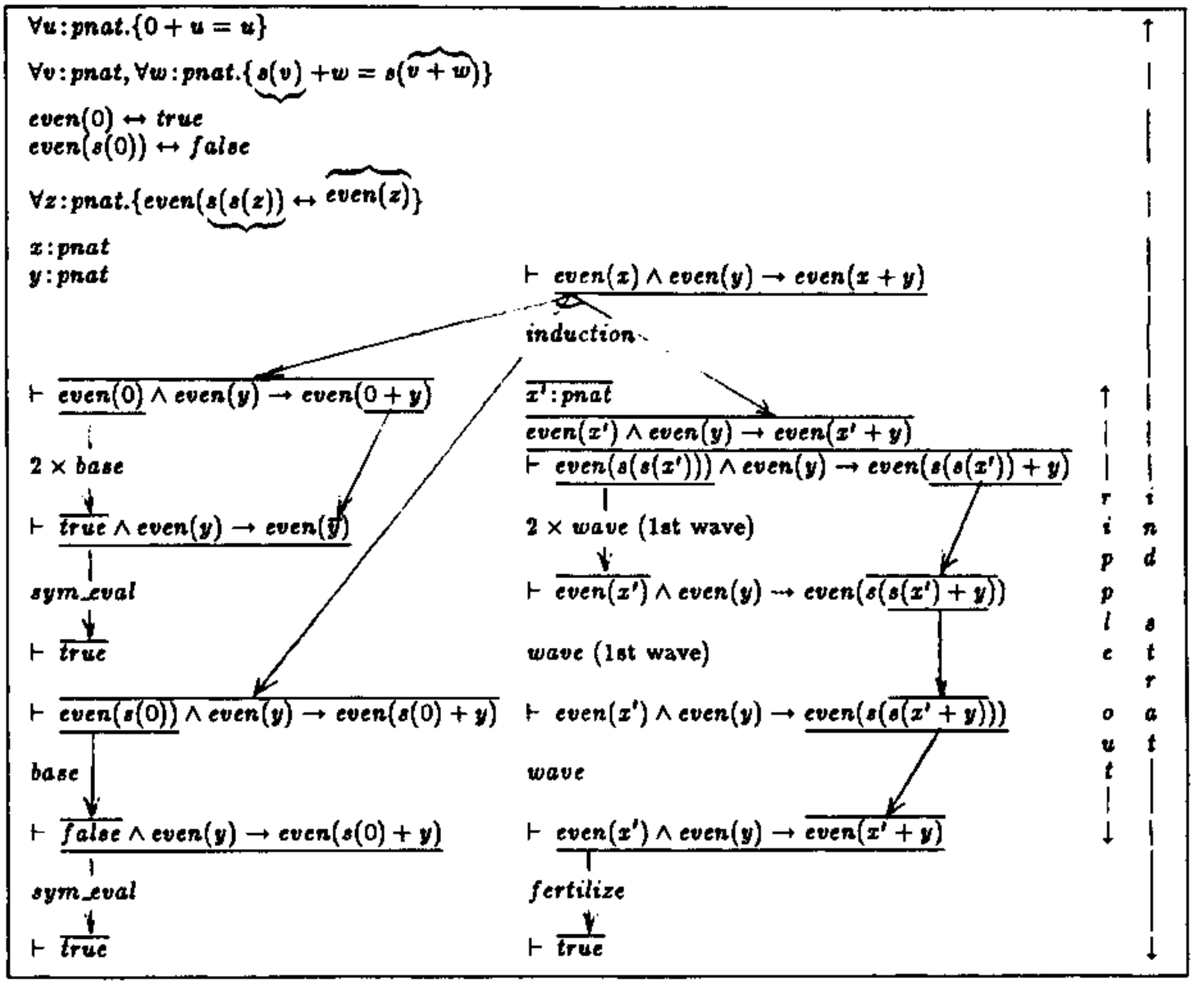
∀u:pnat.{0 + u = u}
∀v:pnat, ∀w:pnat.{s(v) + w = s(v + w)}
even(0) ↔ true
even(s(0)) ↔ false
∀z:pnat.{even(s(s(z))) ↔ even(z)}
x:pnat
y:pnat
⊢ even(x) ∧ even(y) → even(x + y)
induction
⊢ even(0) ∧ even(y) → even(0 + y)
2 × base
⊢ true ∧ even(y) → even(y)
sym_eval
⊢ true
⊢ even(s(0)) ∧ even(y) → even(s(0) + y)
base
⊢ false ∧ even(y) → even(s(0) + y)
sym_eval
⊢ true
x′:pnat
even(x′) ∧ even(y) → even(x′ + y)
⊢ even(s(s(x′))) ∧ even(y) → even(s(s(x′)) + y)
2 × wave (1st wave)
⊢ even(x′) ∧ even(y) → even(s(s(x′) + y))
wave (1st wave)
⊢ even(x′) ∧ even(y) → even(s(s(x′ + y)))
wave
⊢ even(x′) ∧ even(y) → even(x′ + y)
fertilize
⊢ true
ripple out
ind strat

Figure 1: Outline Proof of the Even+ Theorem

3 A Proof Plan for Inductive Proofs

We believe that the BMTP succeeds so often because many inductive proofs have the same overall shape. The BMTP heuristics combine to produce proofs of this overall shape. In particular, the final induction suggestions of recursion analysis initiate proofs of this type and generate subgoals that the other heuristics are designed to solve. We believe that *proof plans* provide a good notation for describing this overall shape and accounting for the way that the heuristics combine together, [Bundy, 1988]. Proof plans can be used to predict and account for the successes and failures of the BMTP. They also suggest ways of improving and extending the BMTP heuristics. In particular, we will show how to use proof plans to account for the success of recursion analysis, and to extend it to make it independent of the forms of recursion found in the conjecture.

We can best explain the overall shape of BMTP proofs by example. Figure 1 is an outline of the proof of conjecture (1) which is generated by our Oyster program using our rational reconstruction of the BMTP heuristics. The Oyster notation has been slightly simplified for expository reasons.

Each formula is a *sequent* of the form *H I-* G, where I-separates the list of *hypotheses, H,* from the *goal, G.* The first sequent is a statement of the conjecture. Each successive sequent is obtained by rewriting a subexpression in the one above it. The subexpression to be rewritten is underlined and the subexpression which replaces it is overlined. Only newly introduced hypotheses are actually written in successive sequents; sequents are to be understood as inheriting all those hypotheses above them in the proof. In the spaces between the sequents are the names of the tactics which invoke the rewriting.

The proof is by backwards reasoning from the statement of the conjecture. The *induction* tactic applies the 2 step induction schema to the conjecture: replacing *x* by 0 and 5(0) in the two base cases, and by *x'* in the induction hypothesis and s(s(x')) in the induction conclusion of the step case. The *base* and *step* tactics then rewrite the base cases and the step case, respectively, using the base and step formulae of the recursive definitions of + and *even.* The two applications of *base* rewrite both the base cases to tautologies, which the *sym_eval* tactic reduces to *true.* The four applications of *step* raise the occurrences of the successor function, s, from their innermost positions around the x's until they are absorbed by the application of the step formula of *even.* The induction conclusion is then identical to the induction hypothesis. The *fertilize* tactic uses the latter to deduce the former.

ripple.out is a super-tactic which is responsible for the repeated application of *step* to the induction conclusion — raising the recursion terms, *ind_strat,* which is an abbreviation of *induction strategy,* is a super-super-tactic for guiding the whole of this proof, apart from the two *sym_eval* applications. It is defined by combining the sub-tactics *induction, base, ripplejout* and *fertilize* in the order suggested by the proof in figure 1. *sym_eval,* which is an abbreviation of *symbolic evaluation,* puts expressions in normal form by unpacking definitions, eliminating quantifiers, applying reflexivity and recognising tautologies. It is also a super-tactic which repeatedly applies *step, base, identity, intro* and *tautology.* A large number of inductive proofs can be understood as a combination *of ind_strait* and *sym_eval,* although not always in the combination illustrated in figure 1. For instance, there are often several nested applications of *ind_strat* before *sym_eval* is used to finish off the proof.

Note how *ripple_out* acts as a bridge between the *induction* and *fertilise* tactics, *induction* makes the induction conclusion differ from the induction hypothesis by the insertion of induction terms for the induction variable. These induction terms form a barrier to *fertilizers* use of the induction hypothesis to prove the induction conclusion, *ripplejout* removes this barrier with a series of *steps* which raise the induction terms out of their inner-most positions. Recursion analysis ensures that the induction terms inserted will have just the right form for the first wave of *steps*[7] to succeed, *i.e.* those *steps* that raise the induction terms past the inner-most functions that dominates them. The raw induction suggestion made by each recursive function occurrence proposes an induction term which is a syntactic variant of that recursive function's recursion term. So if this induction term is inserted then *step* is guaranteed to succeed on this occurrence. If the final induction suggestion is unflawed and subsumes all the raw suggestions, then *step* is guaranteed to succeed on all the occurrences, although repeated *steps* may be required to completely lift the final induction term. This is the reason for choosing an induction schema based on the recursive schemata used to define the functions in the conjecture. These conditions only guarantee the success of this first wave of steps, not the whole *ripple_out* tactic.

4 Fitting Proof Plans to Conjectures

We have implemented each of the tactics illustrated above by a Prolog program which can guide the Oyster proof checker to make the appropriate logical steps. Furthermore, we have specified each of these tactics with a method. CLAM uses these methods to build a tactic tailor-made to prove the whole of each conjecture.

A method is represented as a frame of 6 slots which between them describe the conditions under which its tactic is applicable and the effect if this application is successful.

All that will concern us in this paper are the input formula slot and the preconditions slot of the *ind_strat* method. Consider the preconditions of the version of the *ind_strat* method described in an early version of our program. With some minor generalisations, these were:

$$exp\_at(Conj, [N|Posn]) \equiv X \quad \wedge$$
$$exp\_at(Conj, Posn) \equiv FTerm \quad \wedge$$
$$recursive(FTerm, X, RTerm) \qquad (3)$$

where *Conj* is the input formula, *X* ranges over object-level variables, *FTerm* ranges over object-level terms, *RTerm* ranges over object-level recursion terms, *N* is a natural number and *Posn* is a position. These meta-variables are all universally quantified globally[8] to the method. A position is a list of argument numbers, *e.g.* [1,2,3] would represent the 1st argument of the 2nd argument of the 3rd argument of an expression. *exp_at{Exp, Posn) = SubExp* means that *SubExp* is the subexpression found at position *Posn* in expression *Exp. recursive(FTerm, X, RTerm)* means *FTerm* is a term whose dominent function is recursively defined with the variable, *X,* in its recursive argument and with recursion term, *RTerm, e.g.* the information that *even* is defined recursively with recursive term s(s({/)) is represented by the meta-assertion *recursive(even(U), U, s(s[U)))*[9]. Recursion terms contain the crucial information that CLAM needs to pick an induction schema from those available to Oyster. The final induction suggestion will consist of an recursion term and a recursion argument. CLAM picks an induction schema with an induction term that matches this recursion term. Induction schemata are indexed by their induction terms.

Part of the process of planning involves *fitting* methods to the current conjecture. To fit a method, the input formula is matched to the conjecture and then the preconditions are checked to see if they are satisfied and to instantiate their meta-variables. For instance, fitting these preconditions to the conjecture, (l), above, produces the variable instantiations given in table 2. Comparison of these with table 1 will show that what this method fitting process does is to generate the raw induction suggestions.

X	$FTerm$	$Posn$	$RTerm$
x	$even(x)$	$[1,1,1,1]$	$s(s(x))$
y	$even(y)$	$[2,1,1,1]$	$s(s(y))$
x	$x+y$	$[1,2,1,1]$	$s(x)$

Table 2: Fitting the Original *ind_strat* Method to the Even+ Example

5 Extending Plan Fitting to Perform Recursion Analysis

Two questions arise naturally.

- The preconditions, (3), above only guarantee the success of one application of the *step* tactic for one occurrence of *X.* Can they be extended to at least guarantee the success of the first wave of *steps'!*
- The fitting of the preconditions, (3), above only generates raw induction suggestions. Can they be extended to do recursion analysis, *i.e.* to generate final induction suggestions?

[7] **In figure 1 the first wave of *steps* are marked.**

[8] So the quantifiers do not appear within the preconditions slot, (3), above.

[9] Note the use of the meta-variable *U* ranging over object-level variables, which will be instantiated to whatever *X* is instantiated to when *recursive(FTerm,X, RTerm)* and *recursive(even(U),U, s(s(U)))* are unified.

The answer to both these questions is yes. The beautiful part of these answers is that the same extensions to the preconditions suffice in each case, *i.e.* in extending the preconditions to guarantee the success of later parts of the plan, the processes of flaw checking, subsumption checking and merging are generated naturally as a side-effect. Their presence and success in BMTP is thereby explained.

The extended preconditions are:

$$\begin{array}{ll}
\exists Posn' : lists(pnat). & \\
\quad \{exp\_at(Conj, Posn') \equiv X\} & \wedge \\
\forall Posn' : lists(pnat), \forall N' : pnat. & \\
\{ \quad exp\_at(Conj, [N'|Posn']) \equiv X & \longrightarrow \\
\exists FTerm' : terms, \exists RTerm' : terms. & \\
\{ \quad exp\_at(Conj, Posn') \equiv FTerm' & \wedge \\
\quad recursive(FTerm', X, RTerm') & \wedge \\
\quad\quad subsumes(RTerm, RTerm') & \}\} \qquad \{4)
\end{array}$$

where *subsumes (RTerm, RTerm!)* means that *RTerm* subsumes *RTcrm*[1], *e.g.* *subsumes(s(s(x)),s(x)).* Only those meta-variables whose scope is restricted to the preconditions are quantified within them. The others, *e.g.* *RTerm,* are universally quantified globally to the method.

These revised preconditions extend the original preconditions to every occurrence of every variable, *X,* in the conjecture, *Conj.* They check firstly that each such occurrence is at the recursive argument position of a recursive function and secondly that there is a term, *RTerm,* that subsumes all the recursion terms of these recursive functions. The instantiations of *RTerm* and *X* constitute the final induction suggestions and the instantiations of *RTerm*[1] and *X* the raw induction suggestions. Care has been taken in our implementation so to define *subsumes* that attempts to satisfy it, with its second argument instantiated and its first argument uninstantiated, efficiently produce minimal subsuming schemata.

Fitting these preconditions to the conjecture 1 above, produces the variable instantiations given in table 3. The first line of this table is the required final induction suggestion.

X	$RTerm$	Status
x	$s(s(x))$	unflawed-compatible
y	$s(s(y))$	flawed-compatible

Table 3: Fitting the New *ind_strat* Method to the Even+ Example

The second line of the table is a suggestion that actually /at/5 the preconditions, (4). We have included it and the Status column for the sake of the following discussion. An induction suggestion that meets *all* the preconditions may not always be available, *e.g.* both *x* and y are flawed in x + y = y + x. In such cases we may want to follow BMTP and settle for the suggestion that meets *most* of them. We can classify the variables in the conjecture as follows. Those variables which occur only in recursive positions will be called *unflawed* and those that occur in non-recursive positions will be called *flawed.* Those variables for which there is a schema that subsumes all the schemata of its recursive occurrences will be called *compatible* and the rest will be called *incompatible.* Thus only unflawed-compatible variables completely meet the preconditions (4). Note that an instantiation of *RTcrm* can only be given for compatible variables. Suggestions are preferred in the order: unflawed-compatible, flawed-compatible, unflawed-incompatible, flawed-incompatible. Ideally, our method fitting process would be flexible and accept partially fitting methods if fully fitting ones are not available. However, in the current implementation we have adopted the short term expedient of building this flexibility into the preconditions themselves. See [Bundy *et al.,* 1988] for details.

6 The Use of Lemmas in Rippling-Out

The *step* tactic, which is restricted to using rewrite rules based on the step formula of recursive definitions, is not always strong enough to enable *ripple_out* to succeed. Consider the following expression:

$$even(\underline{s(x) \times y})$$

where x is defined by:

$$\begin{array}{rcl}
\forall v : pnat.\{0 \times v & = & 0\} \\
\forall u : pnat, \forall v : pnat.\{s(u) \times v & = & u \times v + v\}
\end{array}$$

step applies to this once, to produce the expression:

$$\underline{even(\overline{x \times y + y})}$$

but the argument of *even* is not of the form *s(s(*...)), so *step* will not apply a second time; *ripple_out* will fail What would permit another ripple, moving *even* adjacent to *x* X y, is the application of theorem (1) as a rewrite rule[11], *i.e.* applying:

$$even(\underbrace{u + v}) \Rightarrow \overbrace{even(u)} \wedge \overbrace{even(v)} \qquad (5)$$

produces the expression:

$$\overline{even(x \times y) \wedge even(y)}$$

which contains *even(x* x *y)* as required.

In general, we want to extend *step* to a new tactic *wave,* which uses rewrite rules of the form:

$$F(\underbrace{B(U)}) \Rightarrow C(\overbrace{F(U)}) \qquad (6)$$

where *U* ranges over object-level variables, *F* ranges over object-level functions and *B* and *C* range over object-level terms. We will call rewrite rules of the form (6) *wave rules,* where *F(U)* is the *wave function, U* is the *wave argument* and *B(U)* is the *wave term* by analogy to recursive function, recursion argument and recursion term, respectively. The wave terms in wave rules are

[10] This definition of flawing corresponds in spirit to the Boyer-Moore definition, but differs from it in minor ways.

[11] Recall that the proof is constructed by backwards reasoning, so that an implication of the form *P* —► *Q* should be used as a rewrite rule *Q* => *P.*

underbraced and the wave functions are overbraced. The braces in rule (5) above show that it is a wave rule in two ways: one for each occurrence of the wave function on the right hand side of the rule. The *wave* tactic allows us to ripple out through non-recursive functions, non-recursive argument positions of recursive functions, and recursive argument positions that do not contain the appropriate constructor functions.

Step formulae of recursive definitions are usually wave rules. In addition, each theorem that Oyster proves can be tested to see if it has the right form and, if so, it can be stored as a wave rule for future use.

7 Extending Recursion Analysis

Now that we have generalised *ripple_out* beyond the use of step formulae of recursive definitions, it is natural to consider generalising the preconditions of the *ind_strat* method to meet the new opportunities this provides.

The key observation is that the induction term does not have to be derived from just the recursion terms of recursive functions in the conjecture. More generally, it can be derived from some wave rules' wave terms. We require only that its insertion would cause those wave rules to match subexpressions of the induction conclusion. The recursion terms of recursive functions *are* candidates, but they are not the only ones.

To illustrate this, consider the conjecture that every natural number can be factorized into a product of primes. We will formalise this as:

$$\begin{array}{l} \forall x{:}pnat. \\ \quad x \neq 0 \ \rightarrow \ \exists xl{:}lists(primes).\{prod(xl) = x\} \end{array} \quad (7)$$

where *prod* is defined by:

$$\begin{array}{rcl} & prod([\,]) & = & s(0) \\ \forall hd{:}pnat, \forall tl{:}lists(pnat). & & & \\ & prod([hd|tl]) & = & hd \times prod(tl) \end{array}$$

Note that this definition is not based on a *pxx'* recursion.

Now consider the rewrite rule:

$$prod([hd|tl]) = \underbrace{hd \times u} \ \Rightarrow \ \overbrace{prod(tl) = u} \quad (8)$$

As indicated by the braces, it is a wave rule whose wave function is *prod(tl)* = u and whose wave term is *hd x u. prod(xl)* — *x* in the conjecture (7) matches this wave function and has a variable, x, in the wave argument position. If *induction* used the induction term, *p x* x', suggested by this wave term then the wave rule would ripple the induction term out one wave. This suggests a form of induction that substitutes *pxx'* for x, in wnich *x'* is the induction variable. The *pxx'* form of induction:

$$\frac{\Gamma \vdash P(0) \quad \Gamma \vdash P(s(0)) \qquad \Gamma, p{:}primes, x'{:}pnat\ , P(x') \vdash P(\underbrace{p \times x'})}{\Gamma, x{:}pnat \ \vdash P(x)}$$

has just this property. The only other raw induction suggestion that CLAM can extract by matching available wave rules to this occurrence of x, is the very similar, and also successful, x' x x" schema. There are no other occurrences of x in conjecture (7), so no other raw induction suggestions to consider.

Using this *p x x'* induction the first wave of rippling out is done with the wave rule (8), as expected. The remaining rippling out is done with some standard logical manipulation rules. The induction conclusion is then identical to the induction hypothesis, so *fertilize* finishes the proof of the step case.

The generalisation of the preconditions, (4), of *ind_strat* required in order to use wave rules to suggest inductions is:

$$\begin{array}{lll} \exists Posn{:}lists(pnat). & & \\ \quad \{exp\_at(Conj, Posn) \equiv X\} & \wedge & \\ \forall Posn'{:}lists(pnat), \forall N'{:}pnat. & & \\ \{ \quad exp\_at(Conj, [N'|Posn']) \equiv X & \longrightarrow & \\ \exists FTerm'{:}terms, \exists B'{:}terms. & & \\ \{ \quad exp\_at(Conj, Posn') \equiv FTerm' & \wedge & \\ \quad wave\_rule(FTerm', X, B') & \wedge & \\ \quad\quad subsumes(B, B') & \}\} & \quad (9) \end{array}$$

where *wave_rule [FTerm!*, X, *B')* means that there is a wave rule with wave function, *FTerm* wave argument, X, and wave term, *B', e.g.* since (8) is a wave rule with wave function *prod(UL)* = U, wave argument *U* and wave term *Hd x U*, then *wave_rulelprod(UL)* = *U,U,HdxU)* (cf. *recursive {even{U),U,s{s(U)))).*

These preconditions check that each occurrence of *X* is in the an argument position of a function for which there is a matching wave rule, and that there is a term, 5, that subsumes each of the wave terms, B', of these rules. *B* then determines the induction schema and *X* is the induction variable.

If the only wave rules known to CLAM are step cases of recursive definitions then preconditions (9) generate exactly the same final induction suggestions as preconditions (9). If CLAM-Oyster proves theorems that have the form of wave rules and if the existence of these rules is recorded by *wave_rule* meta-assertions then additional induction suggestions are possible. However, in practice, the probability that a wave rule will match each conjecture is quite low, so only a few final induction suggestions will be made even when a large number of wave rules and induction schemata are available. Several of these few suggestions may succeed — leading to different proofs of the conjecture — so that some of the increase in search space is benign. Our experience so far is that the new preconditions substantially increase the number of conjectures for which CLAM can find proof plans, without substantially increasing the size of the search space[12]. This situation may change as the number of wave rules increases, but there is no reason at present to suppose that it will.

8 Implementation and Results

The ideas discussed in this paper have been implemented in and tested on the CLAM-Oyster system. This required the following modifications.

- The only primitive induction schemata provided in Oyster are 1 step schemata for natural numbers,

[12] Although search time does increase, since non-matching wave rules must be tried and rejected.

integers and lists, *i.e.* the $s(x')$, $s(x')\&-x'$, and $[hd|x']$ schemata. We have added a number of new induction schemata, including $s(s(x))$, $p\times x'$, $x'\times x''$ and $x'+x''$ schemata. To ensure the soundness of each new schema it was proved interactively as a higher order theorem in Oyster. This theorem was then available as a lemma for subsequent use by the *induction* tactic. For instance, in order to be able to use the $s(s(x'))$ schema, (2), we had first to prove the theorem:

$$\begin{aligned} &\forall P : pnat \mapsto boole. \\ &P(0) \wedge P(s(0)) \wedge \forall x' : pnat. \{P(x') \rightarrow P(\underbrace{s(s(x'))}) \} \\ &\qquad\qquad \rightarrow \quad \forall x : pnat.\ P(x) \end{aligned}$$

- The Oyster tactics and methods have been upgraded to use these more general forms of induction. The *step* tactic and method have been replaced with the wave ones.
- We have added a new tactic, *existential,* for instantiating existentially quantified variables. It substitutes a meta-variable for the existential variable when eliminating the existential quantifier during the planning process. Unification then instantiates this to an appropriate object-level term as the planning process progresses, *e.g.* during rippling-out. When the *ind_strat* tactic is applied this object-level term is substituted for the existential variable at the time of existential quantifier elimination.
- The preconditions of the *ind_strat* method have been upgraded to those given in meta-formula (9) above, but with modifications to permit partial fitting where total fitting is not possible.
- Plans and proofs for the even+ theorem, (1), and the prime factorization theorem, (7), have been automatically generated. Fitting the new preconditions of the *ind_strat* generates the recursion analysis described above for each of these two examples.
- The improvements in the methods and tactics has enabled simpler proofs to be found for some theorems. For instance, the proof of the commutativity of times, x x y = yx i, which previously required 7 nested applications of *s(x)* induction, now only requires 3 nested inductions, 2 of which are x^1 + x'' inductions.

9 Conclusion

We have analysed recursion analysis by showing how the inductive proofs obtained by the Boyer-Moore Theorem Prover have a common overall shape, and how recursion analysis selects an induction schema and variable that increases the chances that a proof of this shape will be found. We have shown how to describe this overall shape using the technique of proof plans and, hence, both how to rationally reconstruct recursion analysis within proof plans and how to explain its success formally. Our analysis has suggested a natural extension to recursion analysis which frees the choice of induction schema from the forms of recursion present in the original conjecture. In fact, recursion need not be present at all for induction to be used. This is vital if we are to specify programs without making any premature procedural committment, *e.g.* by anticipating the form of recursion to be used in the program, and yet have this procedural commitment made during the synthesis process.

The importance of this rational reconstruction of recursion analysis is illustrated by the following points.

- The neat way that it relates flaw checking, subsumption checking and merging as different aspects of fitting the same preconditions.
- The way in which it suggests an interesting extension of recursion analysis, enabling our theorem prover to find proofs that are beyond the capacity of BMTP.

We have implemented a theorem proving system, CLAM-Oyster, based on our rational reconstruction and shown that this system can find proofs which are beyond the ability of BMTP. Whether this success is brought at the price of an unacceptable increase of the planning search space it is not possible to tell without further empirical testing. The initial empirical results reported in [Bundy *et a/.,* 1988] are very encouraging, however. The planning search space on simple examples is several orders of magnitude smaller than the object-level search space, and the ratio improves with more complex theorems.

The research reported above is in the context of recursive functions with a single recursion variable and a single step formula. This has served to simplify the discussion but the underlying ideas are not limited to such functions. We plan to extend the tactics and methods to more general recursive functions. Only then will we be able to test CLAM-Oyster on the full list of conjectures proved by the BMTP and similar theorem provers.

References

[Boyer and Moore, 1979] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[Bundy *et a/.,* 1988] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. Research Paper 413, Dept. of Artificial Intelligence, Edinburgh, 1988. Submitted to JAR.

[Bundy, 1988] A. Bundy. The use of explicit plans to guide inductive proofs. In *9th Conference on Automated Deduction,* pages 111-120. Springer-Verlag, 1988. Longer version available as DAI Research Paper No. 349.

[Constable *et a/.,* 1986] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Horn, 1988] C. Horn. The Nurprl proof development system. Working paper 214, Dept. of Artificial Intelligence, Edinburgh, 1988. The Edinburgh version of Nurprl has been renamed Oyster.

[Stevens, 1988] A. Stevens. A rational reconstruction of Boyer and Moore's technique for constructing induction formulas. In Y. Kodratoff, editor, *The Proceedings of ECAI-88,* pages 565-570. European Conference on Artificial Intelligence, 1988. Also available as DAI Research Paper No. 360.